

Project Documentation

Integrated Development Environment

Integrated Development Environment (IDE) is the daily-used coding tool for a programmer which enables a complete set for Source Code Editor as well as debugging featured building tool. Over the last few years, Python has emerged as one of the most used languages by the programmers. **Colaboratory**, or “Colab” for short, is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education. Colab is a hosted Jupyter notebook service that requires no setup to use, while providing free access to computing resources including GPUs. It provides a perfect environment for the **data science** students as it includes data cleaning and transformation, numerical simulation, statistical modelling, data visualisation, and many other features.

Python

Python offers stability flexibility and has wide range of tools and libraries which fulfil the aspirations of AI based projects. Python helps developers be productive and confident about the software they’re building. Advantages of Python include simplicity and consistency, access to great libraries and frameworks for AI and machine learning (ML), flexibility, platform independence, and a wide community. These features add to the overall popularity of the language.

Base Code

Our base code includes importing dataset and rearranging columns for compact training of dataset on visual and social features which are trained on dependent variable namely ‘views168’ i.e. number of views on final day of survey i.e.168.

We will dive straight into the Python code and first explain genetic algorithm based on the fragments of code.

Firstly, we import all required objects as shown in fig 1.

```
[1] import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_score
import random
import matplotlib.pyplot as plt

[3] url = "https://raw.githubusercontent.com/saranshtaneja/genetic_major/master/fb_dataset.csv"

[4] data = pd.read_csv(url)

[5] data
```

Fig 1: Import Required Objects

The Facebook video dataset is used in the regression task of predicting the popularity in terms of number of views. There are 700 numeric and categorical variables which decide the popularity or number of views achieved by a video.

```
[77] y_train = data.iloc[0:500,701].values

[80] x_train = data.iloc[0:500,1:701].values

[84] from sklearn.preprocessing import StandardScaler
sc_x=StandardScaler()
sc_y=StandardScaler()
x=sc_x.fit_transform(x_train)
y=sc_y.fit_transform(y_train.reshape(-1,1))

[85] y=y.ravel()
```

Fig 2: Obtain Dependent and Independent variables for training

Use Standard Scaler method from sklearn library as shown in fig 2 which converts respective dataset values to their respective Z values (Standard score) in other words limit the dataset in a confined range to make training the regression model easier and compact.

```
[82] from sklearn.model_selection import GridSearchCV

[83] parameters = {'epsilon':[0,2], 'C':[1, 1000]}
```

Fig 3: Deciding best parameters for training on SVR estimator

Method named GridSearchCV from scikit learn library cross validates and helps decide best parameters to choose for training the proposed support vector regression method.

```
[88] from sklearn.svm import SVR

[89] svr = SVR(kernel='rbf',C = 10000)

[90] clf = GridSearchCV(svr, parameters)

[91] clf.fit(x, y)
      GridSearchCV(estimator=svr,
                    param_grid={'C': [1, 10000], 'epsilon': [0,2]})

➤ GridSearchCV(cv=None, error_score=nan,
               estimator=SVR(C=10000, cache_size=200, coef0=0.0, degree=3,
                             epsilon=0.1, gamma='scale', kernel='rbf',
                             max_iter=-1, shrinking=True, tol=0.001,
                             verbose=False),
               iid='deprecated', n_jobs=None,
               param_grid={'C': [1, 10000], 'epsilon': [0, 2]},
               pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
               scoring=None, verbose=0)

[92] clf.best_params_

➤ {'C': 1000, 'epsilon': 0}
```

Fig 4: Estimating best parameters for SVR model

```
[126] score = np.mean(cross_val_score(svr, x, y, cv=5, scoring=None))
      score

➤ 0.7214387780291002
```

Fig 5: Score for base model

The support vector regression with radial basis function which is based on RBF kernel uses two main parameters, gamma and C that are related to the decision region (how spread the region is), and the penalty for misclassifying a data point as shown in Fig 4. We obtain a score of .72 i.e. approximately 72 percent after taking mean of 5 different validation scores on respective training and test dataset as shown in fig 5.

1.1 Genetic Algorithm

Genetic algorithm consists of different operations and parameters, thus it is beneficial to enclose it as a Python class. We initialize GeneticSelector class with parameters describing genetic algorithm as shown in fig 6.

```
[127] class GeneticSelector :  
  
    def __init__(self, estimator, n_gen, size, n_best, n_rand,  
                n_children, mutation_rate):  
        # Estimator  
        self.estimator = estimator  
        # Number of generations  
        self.n_gen = n_gen  
        # Number of chromosomes in population  
        self.size = size  
        # Number of best chromosomes to select  
        self.n_best = n_best  
        # Number of random chromosomes to select  
        self.n_rand = n_rand  
        # Number of children created during crossover  
        self.n_children = n_children  
        # Probability of chromosome mutation  
        self.mutation_rate = mutation_rate  
        if int((self.n_best + self.n_rand) / 2) * self.n_children != self.size:  
            raise ValueError("The population size is not stable.")
```

Fig 1.1 GeneticSelector python class

Gene: There are 700 features in the Facebook Video dataset. Using genetic algorithm nomenclature a feature is called a gene. It can be either included (1) or excluded (0) during feature selection process as shown in fig 7.

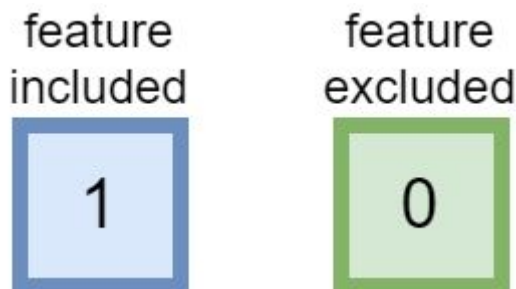


Fig 1.2 Pictorial Representation of Gene

Chromosome: A list of 700 genes is called a chromosome as shown in fig 8. The chromosome contains information which feature is included and which is excluded.

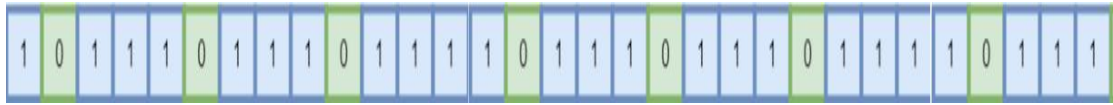


Fig 1.3 Chromosome as a list of 700 genes

Population: It consists of different instances of chromosome as shown in fig 9.

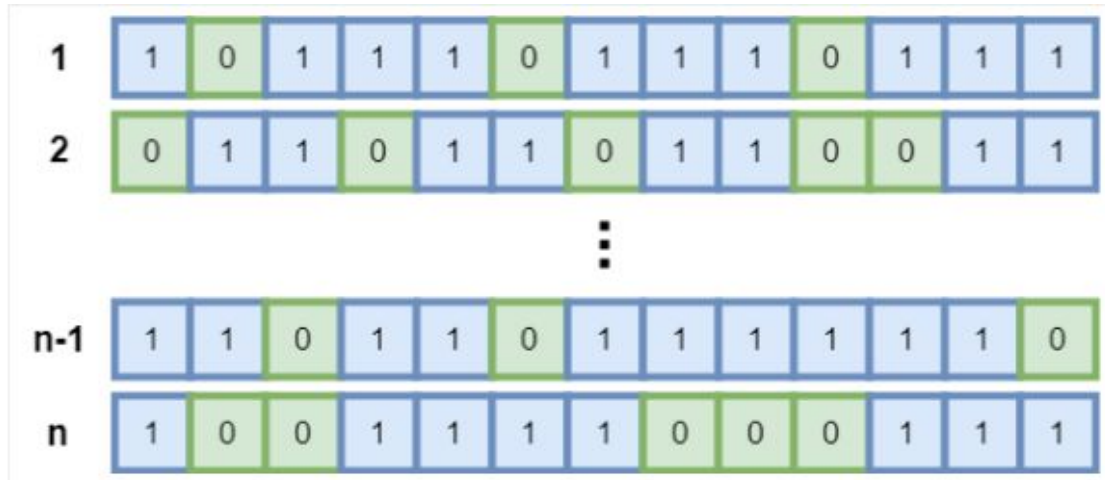


Fig 1.4 A population subset

The first population of ‘n_size’ chromosomes is created by random exclusion of features. We add a method initialize to our class as shown in fig 10.

```
def initilize(self):
    population = []
    for i in range(self.size):
        chromosome = np.ones(self.n_features, dtype=np.bool)
        mask = np.random.rand(len(chromosome)) < 0.3 #The pro
        chromosome[mask] = False
        population.append(chromosome)
    return population
```

Fig 1.5 Initialize method

The probability 0.3 is chosen arbitrarily, however it is suggested to avoid large probabilities. We would not like to create chromosomes with all variables excluded.

Fitness goal is to select such a subset of features that maximizes the SVR accuracy obtained. The function that calculates fitness score for each chromosome in a population is presented in fig 11:

```

def fitness(self, population):
    X, y = self.dataset
    scores = []
    for chromosome in population:
        score = -1* np.mean(cross_val_score(self.estimator, X[:,chromosome], y,
                                            cv=5,
                                            scoring=None))

        scores.append(score)
    scores, population = np.array(scores), np.array(population)
    inds = np.argsort(scores)
    print(-1*scores[inds[0]])
    #print(population[inds[0]])
    return list(scores[inds]), list(population[inds,:])

```

Fig 1.6 Fitness method

The fitness function returns a list of sorted scores and a list of chromosomes sorted based on scores. These two lists will be used in the selection process.

The next step is selection. We select ‘n_best’ chromosomes according to CV scores, so that our population is moving towards the best solution and randomly select ‘n_rand’ chromosomes, so that our optimization algorithm would not be stuck in a local optimum as shown in fig 12.

```

def select(self, population_sorted):
    population_next = []
    for i in range(self.n_best):
        population_next.append(population_sorted[i])
    for i in range(self.n_rand):
        population_next.append(random.choice(population_sorted))
    random.shuffle(population_next)
    return population_next

```

Fig 1.7 Selection method

Next is the ultimate step of crossover where we mix DNA of two individuals. This operation is called crossover and creates ‘n_children’ for each pair of chromosomes.

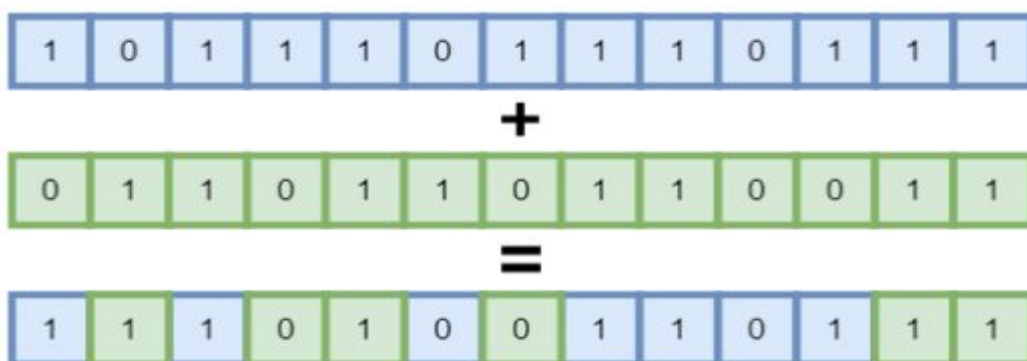


Fig 1.8 Pictorial representation of crossover

We add crossover method to our class, which mixes the genes of the previously selected 'n_best+n_rand' parents as shown in fig 14.

```
def crossover(self, population):
    population_next = []
    for i in range(int(len(population)/2)):
        for j in range(self.n_children):
            #print(i)
            chromosome1, chromosome2 = population[i], population[len(population)-1-j]
            #print(type(chromosome2))
            child = chromosome1
            mask = np.random.rand(len(child)) > 0.5
            for k in range(len(mask)):
                if(mask[k] == True):
                    child[k]=chromosome2[k]
            population_next.append(child)
    return population_next
```

Fig 1.9 Crossover method

The last operation is to mutate chromosomes. A chromosome is changed a little bit in order not to converge to a local optimum too quickly. The change involves randomly excluding a feature with small probability.

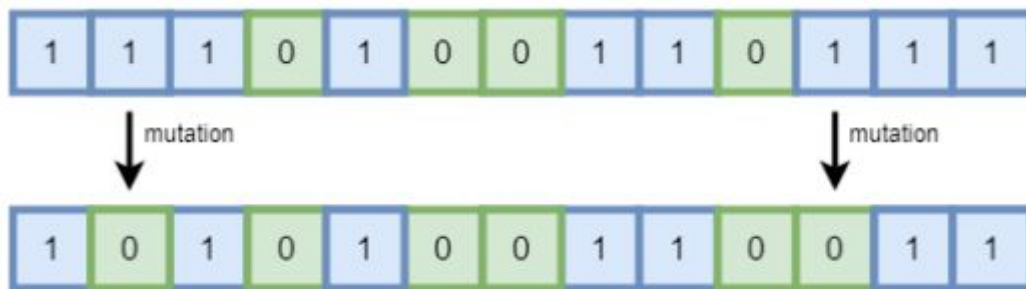


Fig 1.10 Pictorial representation of Mutation

```
def mutate(self, population):
    population_next = []
    for i in range(len(population)):
        chromosome = population[i]
        if random.random() < self.mutation_rate:
            mask = np.random.rand(len(chromosome)) < 0.05
            for k in range(len(mask)):
                if(mask[k]==True):
                    chromosome[k] = False
            population_next.append(chromosome)
    return population_next
```

Fig 1.11 Mutation method

Firstly the chromosome to mutate is randomly selected with probability as 'mutation_rate' and then each gene can be changed with probability .05 as shown in fig 16 These probabilities should not be too large so that genetic algorithm can converge and lead to an effective result.

The genetic operations:

- selection
- crossover
- mutation

The three operations are repeated so that each population should become better and better in terms of the SVR CV scores.

The method named 'generate' calls genetic operations and saves the best results of each generation as shown in fig 17.

```
def generate(self, population):
    # Selection, crossover and mutation
    scores_sorted, population_sorted = self.fitness(population)
    #print(type(population_sorted[1]))
    population = self.select(population_sorted)
    #print(type(population[3]))
    population = self.crossover(population)
    #print(len(population))
    population = self.mutate(population)
    #print(len(population))
    # History
    self.chromosomes_best.append(population_sorted[0])
    self.scores_best.append(scores_sorted[0])
    self.scores_avg.append(np.mean(scores_sorted))
    return population
```

Fig 1.12 Generate method

The last step will be to pass data and perform genetic algorithm. There is also a method that returns a chromosome with the best features (the best chromosome from the last generation) and a plotting function as shown in fig 18.

```

def fit(self, X, y):
    self.chromosomes_best = []
    self.scores_best, self.scores_avg = [], []
    self.dataset = X, y
    self.n_features = X.shape[1]
    population = self.initilize()
    for i in range(self.n_gen):
        population = self.generate(population)
    return self

@property

def support_(self):
    return self.chromosomes_best[-1]

def plot_scores(self):
    plt.plot(self.scores_best, label='Best')
    plt.plot(self.scores_avg, label='Average')
    plt.legend()
    plt.ylabel('Scores')
    plt.xlabel('Generation')
    plt.show()

def ret_pop(self):
    return self.chromosomes_best

```

Fig 1.13 Fit method

Now we've explained all parts of our GeneticSelector class required to perform feature selection. Now apply the genetic algorithm as shown in fig 19 to the Facebook Video dataset and calculate CV score after feature selection.

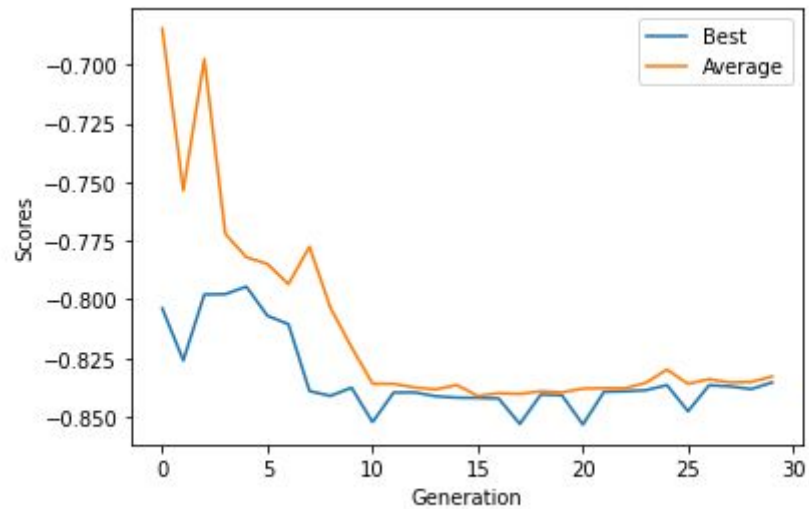
```

selector = GeneticSelector(estimator=svr,
                           n_gen=30, size=100, n_best=20, n_rand=20,
                           n_children=5, mutation_rate=0.05)
selector.fit(x, y)
selector.plot_scores()
score = cross_val_score(svr, x[:,selector.support_], y, cv=5, scoring=None)
print("Score after feature selection: {:.2f}".format(np.mean(score)))

```

Fig 1.14 Applying genetic algorithm for 30 generations

The results obtained after execution of 30 generations are shown as under:



Score after feature selection: 0.84

Fig 1.15 Best and Average Scores

As observed in fig 20 best and average results converge after 26th generation.

We can take a count of features selected in each generation as shown in fig 21

```
for i in range(len(populati)):
    count=0
    for j in range(len(x[0])):
        if(populati[i][j]==True):
            count=count+1
    print("features selected in ",(i+1)," th generation =",count)
```

Fig 1.16 Features selected in each generation code

```

features selected in 1 th generation = 490
features selected in 2 th generation = 472
features selected in 3 th generation = 471
features selected in 4 th generation = 433
features selected in 5 th generation = 454
features selected in 6 th generation = 440
features selected in 7 th generation = 445
features selected in 8 th generation = 403
features selected in 9 th generation = 413
features selected in 10 th generation = 395
features selected in 11 th generation = 392
features selected in 12 th generation = 376
features selected in 13 th generation = 382
features selected in 14 th generation = 381
features selected in 15 th generation = 385
features selected in 16 th generation = 385
features selected in 17 th generation = 368
features selected in 18 th generation = 349
features selected in 19 th generation = 359
features selected in 20 th generation = 348
features selected in 21 th generation = 359
features selected in 22 th generation = 364
features selected in 23 th generation = 354
features selected in 24 th generation = 361
features selected in 25 th generation = 347
features selected in 26 th generation = 340
features selected in 27 th generation = 308
features selected in 28 th generation = 344
features selected in 29 th generation = 337
features selected in 30 th generation = 332

```

Fig 1.17 Features Selected in each Generation

Execution time of genetic algorithm can be a disadvantage in case of more complex dataset and other methods may be more efficient though relatively high number of different feature subsets is checked and the genetic algorithm results on video popularity prediction are proposed henceforth.

2. Particle Swarm Optimization

Particle Swarm optimization is a nature inspired and a meta-heuristic technique that simulates the behavior of bird's flocking. A particle is a set of rows which represent a single bird in the flock. As a bird advances towards destination or we can say that as the particle reaches towards the best score, then rest of the particles follows the leading particle.

As per the algorithm, each particles' position and velocity is updated as it approaches near the best values. The following algorithm explains the mechanism briefly

2.1 Algorithm

```
for each particle  $i = 1, \dots, S$  do
    Initialize the particle's position with a uniformly
    distributed random vector:  $\mathbf{x}_i \sim U(\mathbf{b}_{lo}, \mathbf{b}_{up})$ 
    Initialize the particle's best known position to its
    initial position:  $\mathbf{p}_i \leftarrow \mathbf{x}_i$ 
    if  $f(\mathbf{p}_i) < f(\mathbf{g})$  then
        update the swarm's best known position:  $\mathbf{g} \leftarrow \mathbf{p}_i$ 
    Initialize the particle's velocity:  $\mathbf{v}_i \sim U(-|\mathbf{b}_{up}-\mathbf{b}_{lo}|, |\mathbf{b}_{up}-\mathbf{b}_{lo}|)$ 
while a termination criterion is not met do:
    for each particle  $i = 1, \dots, S$  do
        for each dimension  $d = 1, \dots, n$  do
            Pick random numbers:  $r_p, r_g \sim U(0,1)$ 
            Update the particle's velocity:  $\mathbf{v}_{i,d} \leftarrow \omega \mathbf{v}_{i,d} + \phi_p r_p (\mathbf{p}_{i,d} - \mathbf{x}_{i,d}) + \phi_g r_g (\mathbf{g}_d - \mathbf{x}_{i,d})$ 
```

```
Update the particle's position:  $\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i$ 
```

```
if  $f(\mathbf{x}_i) < f(\mathbf{p}_i)$  then
```

```
    Update the particle's best known position:  $\mathbf{p}_i \leftarrow \mathbf{x}_i$ 
```

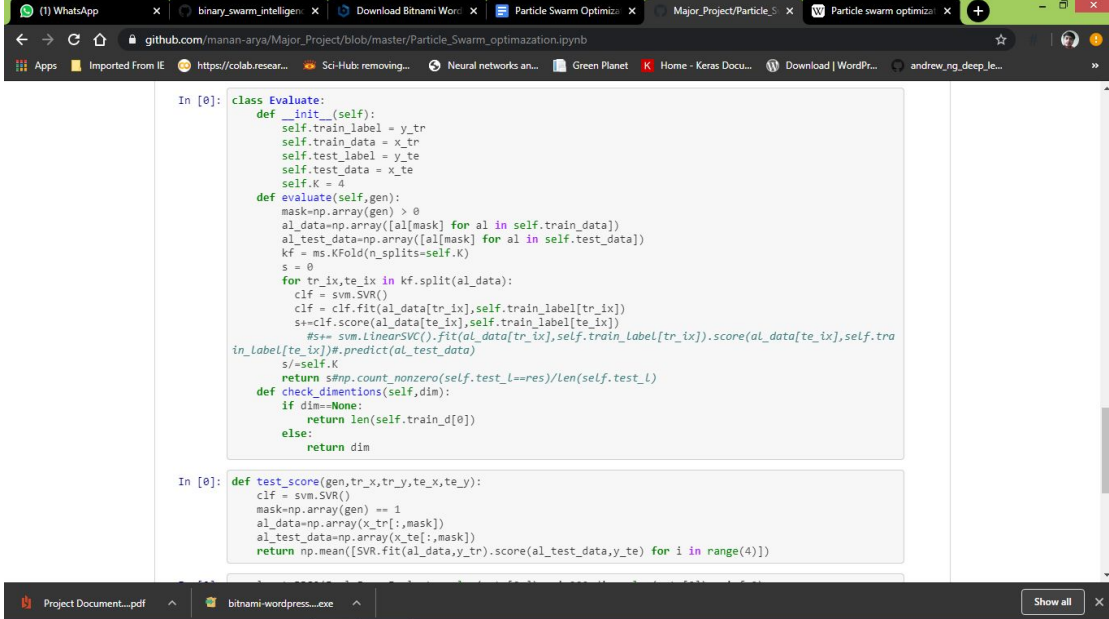
```
if  $f(\mathbf{p}_i) < f(\mathbf{g})$  then
```

```
    Update the swarm's best known position:  $\mathbf{g} \leftarrow \mathbf{p}_i$ 
```

As in computational science, **particle swarm optimization (PSO)** is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. It solves a problem by having a population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formulae over the particle's position and velocity.

Each particle's movement is influenced by its local best known position, but is also guided toward the best known positions in the search-space, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions. So, the class that evaluate the results from PSO nature inspired algorithm having various functions to initialize the variables like training and testing data and there labels. Also, mask array is created to get the impression of selected features those are selected while optimizing.

2.2. Implementation



```
In [0]: class Evaluate:
def __init__(self):
    self.train_label = y_tr
    self.train_data = x_tr
    self.test_label = y_te
    self.test_data = x_te
    self.K = 4
def evaluate(self,gen):
    mask=np.array(gen) > 0
    al_data=np.array([al[mask] for al in self.train_data])
    al_test_data=np.array([al[mask] for al in self.test_data])
    kf = ms.KFold(n_splits=self.K)
    s = 0
    for tr_ix,te_ix in kf.split(al_data):
        clf = svm.SVR()
        clf = clf.fit(al_data[tr_ix],self.train_label[tr_ix])
        s+=clf.score(al_data[te_ix],self.train_label[te_ix])
        #svm.LinearSVC().fit(al_data[tr_ix],self.train_label[tr_ix]).score(al_data[te_ix],self.train_label[te_ix])#predict(al_test_data)
    s/=self.K
    return s#np.count_nonzero(self.test_L==res)/len(self.test_L)
def check_dimensions(self,dim):
    if dim==None:
        return len(self.train_d[0])
    else:
        return dim

In [0]: def test_score(gen,tr_x,tr_y,te_x,te_y):
    clf = svm.SVR()
    mask=np.array(gen) == 1
    al_data=np.array(x_tr[:,mask])
    al_test_data=np.array(x_te[:,mask])
    return np.mean([SVR.fit(al_data,y_tr).score(al_test_data,y_te) for i in range(4)])
```

Fig 2.1 : Evaluate class for implementation of PSO algorithm

PSO is a metaheuristic as it makes few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. However, metaheuristics such as PSO do not guarantee an optimal solution is ever found. Also, PSO does not use the gradient of the problem being optimized, which means PSO does not require that the optimization problem be differentiable as is required by classic optimization methods such as gradient descent. Now, this Binary PSO function is defined as shown below :

```

111     gens.append(alter_gens[1])
112     return best_val,best_pos,best_pos.count(1)
113
114     """BPSO"""
115     def logsig(n): return 1 / (1 + math.exp(-n))
116     def sign(x): return 1 if x > 0 else (-1 if x!=0 else 0)
117
118     def BPSO(Eval_Func,n=20,m_i=200,minf=0,dim=None,prog=False,w1=0.5,c1=1,c2=1,vmax=4):
119         """
120         input:{
121             Eval_Func: Evaluate_Function, type is class
122             n: Number of population, default=20
123             m_i: Number of max iteration, default=300
124             minf: minimization flag, default=0, 0=maximization, 1=minimization
125             dim: Number of feature, default=None
126             prog: Do you want to use a progress bar?, default=False
127             w1: move rate, default=0.5
128             c1,c2: It's are two fixed variables, default=1,1
129             vmax: Limit search range of vmax, default=4
130         }
131
132         output:{
133             Best value: type float 0.967
134             Best position: type list(int) [1,0,0,1,....]
135             Number of 1s in best position: type int [0,1,1,0,1] + 3
136         }
137         """
138         estimate=Eval_Func().evaluate
139         if dim==None:
140             dim=Eval_Func().check_dimensions(dim)
141         pcenter=random_search(n,dim)

```

Fig 2.2 Function definition of PSO

After definition of Evaluate class and the Particle Swarm Optimization function , now the execution statement is called. In this function call Evaluate class is passed from where whole data has to enter the main executing function. According to the outputs, the accuracy of algorithm is calculated and the best selected features are generated in form of a binary array having size equal to that of no. of columns in the dataset.

The calling statement is :

```

s,g,l=opt.BPSO(Eval_Func=Evaluate,n=len(x_tr[0:]),m_i=200,dim =
len(x_tr[0]) ,minf=0)

```

2.3 OUTPUTS

The output generated is as follows:

```
[ ] 1 s # Score
[ ] 0.7618801016101333
[ ] 1 l # no. of features selected
[ ] 240
```

Fig 2.3 : output of PSO algorithm as accuracy and number of selected features

[illegible]

Fig 2.4 :Generated binary array of selected features by PSO algorithm

3. Bat Optimization Algorithm

Common arguments with algorithms:

- Eval_Func: Evaluate function (class)
- n: number of population (int)
- m_i: number of max iteration (int)
- dim: number of all feature (int)
- minf: minimization flag. min or max (bool)

1. random_search function

This function generates and returns a random gene for the first iteration of the algorithm as shown in the figure below: -

```
In [0]: """Common Function"""
def random_search(n,dim):
    """
    create genes list
    input:{ n: Number of population, default=20
           dim: Number of dimension
    }
    output:{genes_list → [[0,0,0,1,1,0,1,...]...n]
    }
    """
    gens=[[0 for g in range(dim)] for _ in range(n)]
    for i,gen in enumerate(gens) :
        r=random.randint(1,dim)
        for _r in range(r):
            gen[_r]=1
        random.shuffle(gen)
    return gens
```

Fig. 3.1 random search function

2. BBA function:

The BBA function acts as the most important function in our implementation of the Bat Algorithm it takes 10 input parameters and outputs 3 values as shown in fig 3.2:

Inputs:

- Evaluate class object
- Population number
- Maximum iteration
- Minimization flag
- Number of features
- Loudness
- Pulse rate

Outputs:

- Best value of the required score in float
- List of features selected
- Number of features selected

```

In [0]: """BBA"""
def BBA(Eval_Func,n=20,m_i=200,dim=None,minf=0,prog=False,qmin=0,qmax=2,loud_A=0.25,r=0.4):
    """
    input:{ Eval_Func: Evaluate_Function, type is class
            n: Number of population, default=20
            m_i: Number of max iteration, default=300
            minf: minimazation flag, default=0, 0=maximization, 1=minimization
            dim: Number of feature, default=None
            prog: Do you want to use a progress bar?, default=False
            qmin: frequency minimum to step
            qmax: frequency maximum to step
            loud_A: value of Loudness, default=0.25
            r: Pulse rate, default=0.4, Probability to relocate near the best position
            }
    output:{Best value: type float 0.967
            Best position: type List(int) [1,0,0,1,.....]
            Nunber of 1s in best position: type int [0,1,1,0,1] → 3
            }
    """
    estimate=Eval_Func().evaluate
    if dim==None:
        dim=Eval_Func().check_dimention(dim)
    #flag=dr
    #qmin=0
    #qmax=2
    #loud_A=0.25
    #r=0.1
    #n_iter=0
    gens_dic={tuple([0]*dim):float("-inf") if minf == 0 else float("inf")}
    q=[0 for i in range(n)]
    v=[[0 for d in range(dim)] for i in range(n)]
    #cgc=[0 for i in range(max_iter)]
    fit=[float("-inf") if minf == 0 else float("inf") for i in range(n)]
    #dr=False
    gens=random_search(n,dim)#[[random.choice([0,1]) for d in range(dim)] for i in range(n)]

```

Fig. 3.2 BBA function

3. Evaluate Class:

Evaluate function takes the modified feature set and run the SVR algorithm over it. It takes its class object as input and returns the regressor score as shown in the image below: -


```
In [0]: class Evaluate:
def __init__(self):
    self.train_label = y_tr
    self.train_data = x_tr
    self.test_label = y_te
    self.test_data = x_te
    self.K = 4
def evaluate(self,gen):
    mask=np.array(gen) > 0
    al_data=np.array([al[mask] for al in self.train_data])
    al_test_data=np.array([al[mask] for al in self.test_data])
    kf = ms.KFold(n_splits=self.K)
    s = 0
    for tr_ix,te_ix in kf.split(al_data):
        clf = svm.SVR()
        clf = clf.fit(al_data[tr_ix],self.train_label[tr_ix])
        s+=clf.score(al_data[te_ix],self.train_label[te_ix])
    s/=self.K
    return s #np.count_nonzero(self.test_l==res)/len(self.test_l)
def check_dimentions(self,dim):
    if dim==None:
        return len(self.train_d[0])
    else:
        return dim
```

Fig. 3.3 Evaluate Class

4. Final Output Values:

Bs: It is the final score obtained by running the algorithm

Bl: It determines the number of features selected

Bg: It is a numpy array containing the features selected by the algorithm

```
In [0]: Bs,Bg,Bl=BBA(Eval_Func=Evaluate,n=len(x_tr[0:]),m_i=200, dim = len(x_tr[0]) ,minf = 0 )
```

```
In [37]: Bs
```

```
Out[37]: 0.7677831250931786
```

In [38]: B1

Out[38]: 172

```
In [0]: BG = np.asarray(Bg)
```

```
In [42]: BG.reshape(1,700)
```

```
Out[42]: array([[0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1,
0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1,
0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0,
0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1,
0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1,
1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1,
0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0,
0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1,
1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0,
0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0,
1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0])
```

4. Dragonfly Algorithm

1. random_search function

This function generates and returns a random gene for the first iteration of the algorithm as shown in the figure below:

```
In [0]: """Common Function"""
def random_search(n,dim):
    """
    create genes list
    input:{ n: Number of population, default=20
           dim: Number of dimension
          }
    output:{genes_list → [[0,0,0,1,1,0,1,...]...n]
           }
    """
    gens=[[0 for g in range(dim)] for _ in range(n)]
    for i,gen in enumerate(gens) :
        r=random.randint(1,dim)
        for _r in range(r):
            gen[_r]=1
        random.shuffle(gen)
    return gens
```

Fig. 4.1 random search function

2. Evaluate Class:

Evaluate function takes the modified feature set and run the SVR algorithm over it. It takes its class object as input and returns the regressor score as shown in the image below: -

```
In [0]: class Evaluate:
def __init__(self):
    self.train_label = y_tr
    self.train_data = x_tr
    self.test_label = y_te
    self.test_data = x_te
    self.K = 4
def evaluate(self,gen):
    mask=np.array(gen) > 0
    al_data=np.array([al[mask] for al in self.train_data])
    al_test_data=np.array([al[mask] for al in self.test_data])
    kf = ms.KFold(n_splits=self.K)
    s = 0
    for tr_ix,te_ix in kf.split(al_data):
        clf = svm.SVR()
        clf = clf.fit(al_data[tr_ix],self.train_label[tr_ix])
        s+=clf.score(al_data[te_ix],self.train_label[te_ix])
    s/=self.K
    return s #np.count_nonzero(self.test_l==res)/len(self.test_l)
def check_dimentions(self,dim):
    if dim==None:
        return len(self.train_d[0])
    else:
        return dim
```

Fig. 4.2 Evaluate Class

3. BDFA function:

The BBA function acts as the most important function in our implementation of the Bat Algorithm it takes 10 input parameters and outputs 3 values as shown in the figure below:

Inputs:

- Evaluate class object
- Population number
- Maximum iteration
- Minimization flag
- Number of features

Outputs:

- Best value of the required score in float
- List of features selected

- Number of features selected

```
In [0]: """BDFA"""
def BDFA(Eval_Func,n=20,m_i=200,dim=None,minf=0,prog=False):
    """
    input:{ Eval_Func: Evaluate_Function, type is class
            n: Number of population, default=20
            m_i: Number of max iteration, default=300
            minf: minimization flag, default=0, 0=maximization, 1=minimization
            dim: Number of feature, default=None
            prog: Do you want to use a progress bar?, default=False
            }
    output:{Best value: type float 0.967
            Best position: type list(int) [1,0,0,1,....]
            Number of 1s in best position: type int [0,1,1,0,1] → 3
            }
    """
    estimate=Eval_Func().evaluate
    if dim==None:
        dim=Eval_Func().check_dimensions(dim)
    maxiter=m_i#500
    #flag=dr#True
    best_v=float("-inf") if minf == 0 else float("inf")
    best_p=[0]*dim
    gens_dict={tuple([0]*dim):float("-inf") if minf == 0 else float("inf")}

    enemy_fit=float("-inf") if minf == 0 else float("inf")
    enemy_pos=[0 for _ in range(dim)]
    food_fit=float("-inf") if minf == 0 else float("inf")
    food_pos=[0 for _ in range(dim)]

    fit=[0 for _ in range(n)]
    genes=random_search(n,dim)
    genesX=random_search(n,dim)

    if prog:
        miter=tqdm(range(m_i))
    else:
        miter=range(m_i)
    for it in miter:
        w=0.9 - it * ((0.9-0.4) / maxiter)
        mc=0.1- it * ((0.1-0) / (maxiter/2))
        if mc < 0:
            mc=0
```

Fig. 4.3 BDFA function

4. Final Output Values:

Cs: It is the final score obtained by running the algorithm

Cl: It determines the number of features selected

Cg: It is a numpy array containing the features selected by the algorithm


```

In [0]: Cs,Cg,C1=BDFA(Eval_Func=Evaluate,n=len(x_tr[0:]),m_i=200, dim = len(x_tr[0]), minf = 0)

In [44]: Cs # Accuracy
Out[44]: 0.7723612500099544

In [45]: C1 # no. of features
Out[45]: 319

In [0]: CG = np.asarray(Cg)

In [47]: CG.reshape(1,700) # Feature array
Out[47]: array([[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1,
0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1,
1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1,
1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0,
1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0,
1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0,
0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0,
0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0,
0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1,
0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0,
1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0,
1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0,
0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0,
1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1,

```

Fig. 4.4 Output Values